

第2章

LSIの中の基本部品を原理から理解する

――実設計に応用できる演算回路スキルを身につけよう

外村元伸

ここでは，新人エンジニアを対象に，LSIの中で使われる演算器とその回路設計について解説します．本稿では数式がたくさん出てきますが，演算回路を設計する以上，その原理を押さえることはとても重要です．一見，難しそうに見えるかもしれませんが，じっくりと式を追っていけば理解できると思います．新人エンジニアを教育する立場のベテラン・エンジニアの方々にも，もう一度それぞれの知識を見直すという意味で，一読していただければと思います．（筆者）

ディジタル回路において，数値は電気的に実現しやすい‘0’と‘1’の状態を利用して，2進数ベースで表現されます．したがって，回路設計者は10進数ではなく，2進数ですべてを扱うことに慣れる必要があります．2進数表現に慣れてくると，思わぬところで簡単な（そして重要な）規則を発見することがあります．ついでと言ってはなんですが，数値を総和記号Σで表現することにも慣れていただければと思います．

Column　回路設計の「ココがすてき」

演算器にはさまざまな種類があります．加減乗除算器のほかに，開平，対数，三角関数，フーリエ変換器（画像認識，電波処理），離散コサイン変換器（画像圧縮），3次元回転行列演算器（3次元コンピュータ・グラフィックス），補間演算器（画面サイズ変換，画面合成）などがあります．これら演算器は，すでにみなさんが日ごろ使用している電子機器の中になんらかの形で実現されています．しかし，高速化や低消費電力化，および小規模化をねらってまだまだ改良の余地があります．また，応用を考えたとき，その設計法がとてもおもしろく，筆者の興味はつきません．

1 頭の中を2進数に

2進nビットの正の整数は，次式のように表現します．

$$\left.\begin{aligned}&[a_{n-1}a_{n-2}\cdots a_2a_1a_0]b=\sum_{i=0}^{n-1}a_i\cdot 2^i\\&=a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_2 2^2+a_1 2^1+a_0 2^0\end{aligned}\right\}\quad\cdots\cdots(1)$$

式(1)において，係数a_iは‘0’か‘1’で，bはbinary（2進数）であることを表しています．また，mビットの正の小数は式(2)で表せます．

$$\left.\begin{aligned}&[0.a_{-1}a_{-2}\cdots a_{-(m-1)}a_{-m}]b=\sum_{i=1}^{m}a_{-i}\cdot 2^{-i}\\&=a_{-1}2^{-1}+a_{-2}2^{-2}+\cdots+a_{-(m-1)}2^{-(m-1)}+a_{-m}2^{-m}\end{aligned}\right\}\quad\cdots(2)$$

したがって，整数部nビット，小数部mビットの正の固定小数点表現は，式(3)のようになります．

$$\left.\begin{aligned}&[a_{n-1}a_{n-2}\cdots a_2a_1a_0.a_{-1}a_{-2}\cdots a_{-(m-1)}a_{-m}]b\\&=\sum_{i=0}^{n-1}a_i\cdot 2^i+\sum_{i=1}^{m}a_{-i}\cdot 2^{-i}\\&=a_{n-1}2^{n-1}+\cdots+a_1 2^1+a_0 2^0+a_{-1}2^{-1}+\\&\quad\cdots+a_{-(m-1)}2^{-(m-1)}+a_{-m}2^{-m}\end{aligned}\right\}\quad\cdots\cdots\cdots(3)$$

これらの表現に慣れるため，例を挙げてみましょう．

$$1/5=0.2=[0.001100110011001100110011001100\cdots]b$$

という数字を見てみると，“0011”という2進数列が繰り返

KeyWord　2の補数，NOT，AND，OR，XOR，セレクタ論理，ブール関数，ルックアップ・テーブル，フル・アダー，4-2コンプレッサ

2

されている，いわゆる「循環2進小数」になっています．循環2進小数は$[0.\dot{0}\dot{0}\dot{1}\dot{1}]\mathrm{b}$のように表現します．では，3/5はどうでしょうか．これは，

$$3/5 = 0.6 = [0.1001100110011001100110011001100\ldots]\mathrm{b}$$
$$= [0.1\dot{0}\dot{0}\dot{1}\dot{1}]\mathrm{b}$$

となります．つまり，3/5は$[0.1]\mathrm{b}$と1/5＝$[0.\dot{0}\dot{0}\dot{1}\dot{1}]\mathrm{b}$の右1ビット・シフトで表現できます．これは，3/5－1/2＝6/10－5/10＝1/10からもわかるでしょう．

$1/255 = [0.0000000100000001\ldots]\mathrm{b} = [0.\dot{0}\dot{0}\dot{0}\dot{0}\dot{0}\dot{0}\dot{0}\dot{1}]\mathrm{b}$も循環2進小数です．しかも，定数1/255を乗算するときは，乗除算なしにそのままで実現できることに留意してください．

もう一つ，10進→2進変換について考えてみましょう．まず，整数の10進 → 2進変換は式(1)より次式のように変形できます．

$$\sum_{i=0}^{n-1} a_i \cdot 2^i = \left(\cdots\left(\left(a_{n-1}2 + a_{n-2}\right)2 + \cdots + a_2\right)2 + a_1\right)2 + a_0$$

これにより，2で割った余りが順番に係数a_iを決定していくことがわかります．次に，小数の10進→2進変換は式(2)より次式のように変形できます．

$$\sum_{i=1}^{m} a_{-i} \cdot 2^{-i} = \left(\cdots\left(\left(a_{-m}2^{-1} + a_{(-m-1)}\right)2^{-1} + \cdots + a_{-2}\right)2^{-1} + a_{-1}\right)2^{-1}$$

2倍した結果の整数部が順々に係数a_{-i}になることがわかります．

● 2の補数表現 —— 論理反転値＋1がなぜ成り立つか

上述の正の数だけを扱う場合は「符号なし(unsigned)数」を用いますが，ディジタル回路の中では負の数も扱うため，2の補数表現を用いた「符号付き(signed)数」で考えます．ここで，符号付き数は絶対値に符号ビットを付けた表現ではないことに注意しましょう．

符号付き数は減算を加算と同じように扱えるので，減算器と加算器の両方を準備しなくてすみます．ここで，符号付き数$A = [a_{n-1}a_{n-2}\cdots a_2 a_1 a_0]\mathrm{b}$は式(4)のように表せます．

$$\left.\begin{aligned} A &= -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \\ &= -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_2 2^2 + a_1 2^1 + a_0 2^0 \end{aligned}\right\} \cdots(4)$$

－Aの2の補数表現を求めるために次式を用います．

$$\left.\begin{aligned} S &= 2^{n-1} - 1 = \sum_{i=0}^{n-2} 2^i \\ &= 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2^1 + 2^0 = 2S - S \end{aligned}\right\} \cdots\cdots\cdots(5)$$

式(5)を変形すると，

$$0 = \sum_{i=0}^{n-2} 2^i - \left(2^{n-1} - 1\right)$$

なので，これに式(4)を代入して，

$$\left.\begin{aligned} -A &= 0 - \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i\right) \\ &= \sum_{i=0}^{n-2} 2^i - \left(2^{n-1} - 1\right) - \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i\right) \\ &= -\left(1 - a_{n-1}\right)2^{n-1} + 1 + \sum_{i=0}^{n-2}\left(1 - a_i\right) \cdot 2^i \\ &= -\bar{a}_{n-1}2^{n-1} + 1 + \sum_{i=0}^{n-2} \bar{a}_i \cdot 2^i \end{aligned}\right\} \cdots(6)$$

が求まります．$\bar{a}_i$がa_iの論理反転であるとすると，式(6)より－AはAの論理反転＋1で表せます．すなわち，$\bar{A} = [\bar{a}_{n-1}\bar{a}_{n-2}\cdots\bar{a}_2\bar{a}_1\bar{a}_0]\mathrm{b}$とすれば，$-A = \bar{A} + 1$となります．

次に，符号付き数X，Yの乗算($X \times Y$)について考えます．ここで，X，Yは次式で与えられます．

$$X = [x_{n-1}x_{n-2}\cdots x_1 x_0]\mathrm{b} = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \qquad \cdots(7)$$

$$Y = [y_{m-1}y_{m-2}\cdots y_1 y_0]\mathrm{b} = -y_{m-1}2^{m-1} + \sum_{j=0}^{m-2} y_j \cdot 2^j \qquad \cdots(8)$$

乗算$X \times Y$の結果を2の補数表現で得るためには，結果の最上位けたの符号部のみを残し，残りすべてを正項の表現に変換しなければなりません．すなわち，式(7)と式(8)をかけ算してから展開し，負項については式(6)を用いて正項に変換していきます．すると，以下に示す式(9)が得られます．

$$\begin{aligned} X \times Y &= x_{n-1}y_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} x_i y_j 2^{i+j} \\ &\quad - y_{m-1}2^{m-1}\sum_{i=0}^{n-2} x_i 2^i - x_{n-1}2^{n-1}\sum_{j=0}^{m-2} y_j 2^j \\ &= -\left(x_{n-1} + y_{m-1} - x_{n-1}y_{m-1}\right)2^{n+m-2} \\ &\quad + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} x_i y_j 2^{i+j} \end{aligned}$$

$$
\left.
\begin{aligned}
&+\sum_{i=0}^{n-2}\bar{x}_i y_{m-1}2^{i+m-1}+\sum_{j=0}^{m-2}x_{n-1}\bar{y}_j 2^{j+n-1}\\
&+y_{m-1}2^{m-1}+x_{n-1}2^{n-1}\\
=&-\left(x_{n-1}\vee y_{m-1}\right)2^{n+m-2}+\sum_{i=0}^{n-2}\sum_{j=0}^{m-2}x_i y_j 2^{i+j}\\
&+\sum_{i=0}^{n-2}\bar{x}_i y_{m-1}2^{i+m-1}+\sum_{j=0}^{m-2}x_{n-1}\bar{y}_j 2^{j+n-1}\\
&+y_{m-1}2^{m-1}+x_{n-1}2^{n-1}
\end{aligned}
\right\}\cdots\cdots(9)
$$

ここで，2^{n+m-2}の項はけた上げが発生しないように，論理和∨を用いて簡単化します．

後は，式(9)をもとに**図1**に示すように筆算の要領で積項を積み上げればよいのです．そしてけたごとに1の数をカウントし，けたの和とけた上げを効率良く求めます．けた上げを効率良く行う方法については後で詳しく説明しますが，まずは乗算器が筆算の原理で簡単に設計できるということを理解してください．**図1**では，符号を示すビットのけたには，特別にマイナス(－)記号を付加して区別しています．

ソフトウェア(プログラミング言語)では，CPUの命令セットを用いて演算式を書くため，2入力とも符号付き数の演算，または2入力とも符号なし数の演算しか許されないのが普通です．しかし，ハードウェアで演算器をみずから設計するときは，一方が符号付き数で他方が符号なし数という演算も可能となります．とくに限られたビット長で数を扱いたいときに有効です．

なお，上の例からわかるように，「符号付き数×符号付き数」の場合，符号ビットの位置が最上位けたの1ビット右へシフトしています．このように符号付きの固定小数点演算では，小数点位置が1ビット右へシフトするので注意が必要です．数の有効けた数からみれば1けた分もったいない話ですが，下位ビットを捨てる必要がある場合は左に1ビット・シフトしてから下位ビットを捨てます．

2 基本的な演算器とその回路構成

演算器の設計は，NOT(否定．インバータともいう)，AND(論理積)，OR(論理和)，XOR(排他的論理和)，SELECTOR(セレクタ論理)などのゲート論理によって構成する演算器を設計するレベルと，ゲート論理を具体的にどのような電気回路素子の組み合わせ(現在ではほとんどMOSトランジスタ回路)で実現するかというレベルに分かれます．本稿では，前者の設計レベルを「演算器」，後者の設計レベルを「演算回路」として区別して説明します．

● 論理設計の基本を押さえる

ここで，ゲート論理設計の基本を説明しておきます．

n個の{0，1}入力に対して出力{0，1}を割り当てる関数f：{0，1}×{0，1}×…×{0，1}→{0，1}をブール関数といいます．FPGAでは，任意のブール関数をルックアップ・テーブル(LUT)によって実現しています．そのため，ルックアップ・テーブルの各ブール関数の性能は同程度として扱えます．

それに対して，ASICでは基本的にはCMOS回路で論理を構成し，AND(∧)，OR(∨)，NOT(¯)を組み合わせて任意のブール関数を実現します．例えば，XOR($=a\wedge\bar{b}\vee\bar{a}\wedge b$)はAND，OR，NOTの組み合わせによって実現するので，性能が低下することがあります．

図2に示すのは，CMOSで構成するNOT(インバータ)回路，NAND回路，NOR回路などの基本回路です．**図2(a)**のインバータは，入力が‘0’のとき，pMOS側が導通して出力は‘1’になります．逆に，入力が‘1’のときはnMOS側が導通し，出力は‘0’になります．つまり，入力状態が反転して出力されます．これらはゲートの論理機能を実現するという役割だけでなく，論理出力の電圧レベルを回復するという役割ももっています．なお，本稿では

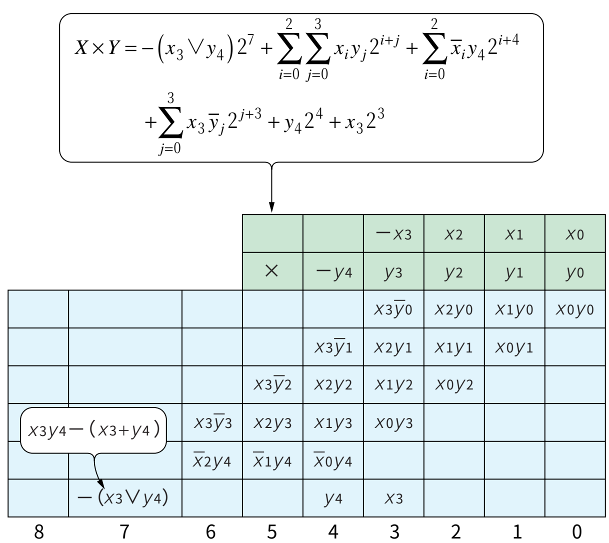
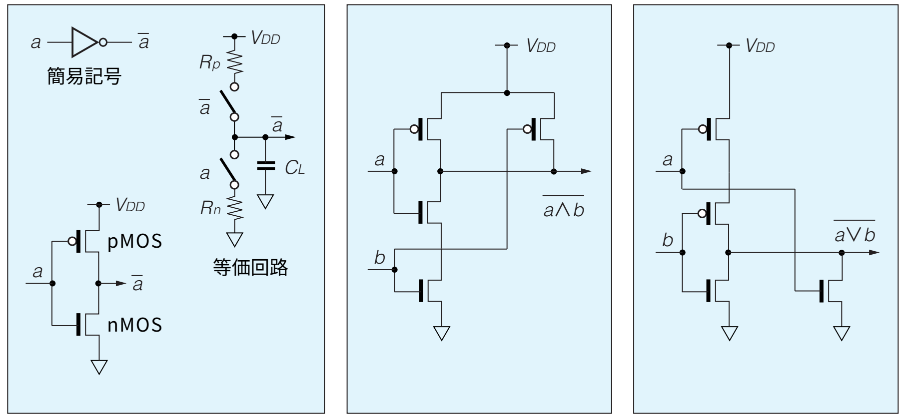


図1　符号付き数×符号付き数の乗算例

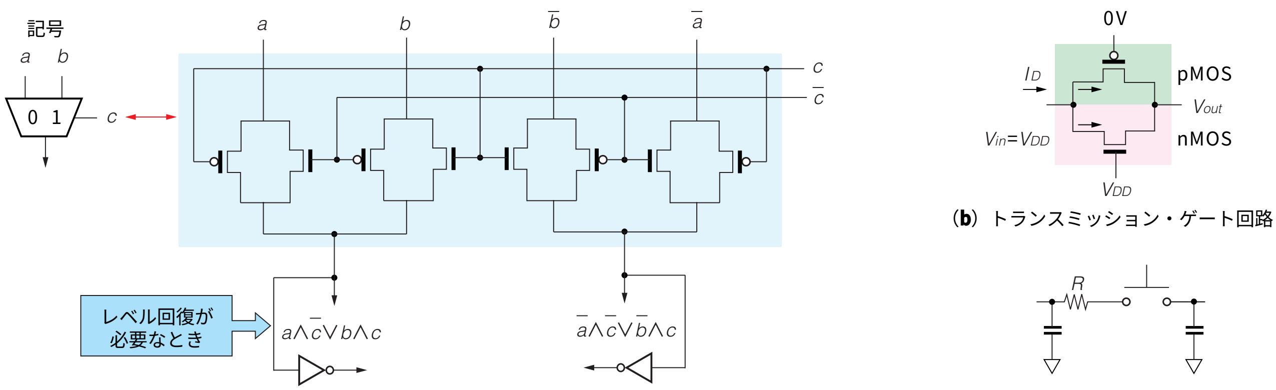
式(9)に$n=4$，$m=5$を代入した例．筆算の要領で，けたごとに積項を書き並べる．

図2
基本ゲート論理を実現するCMOS回路

CMOS回路の基本は，(a)のインバータである．等価回路を見ればわかるように，入力aが反転した$\bar{a}$を出力する．入力を反転すると同時に，動作電圧のレベルも回復する．ANDとORもインバートされた状態（NAND，NOR）で実現される．



(a) インバータ　(b) NAND　(c) NOR



(a) セレクタ論理　(b) トランスミッション・ゲート回路　(c) 等価回路

図3　トランスミッション・ゲート回路で実現するセレクタ論理

セレクタ論理は，基本的にはトランスミッション・ゲート回路で実現できる．実際には，回路の規模や性能に応じて，ダブル・レールかシングル・レールかを使い分ける．電圧降下によるレベル回復の回路挿入も適宜行う．

図2に示した回路記号を使いますが，回路特性の違いを表現するため，いくつか異なる表記が使われる場合もあります．ここでは論理機能のみに注目します（例えば，回路特性を利用した高度な最適化には立ち入らない）．

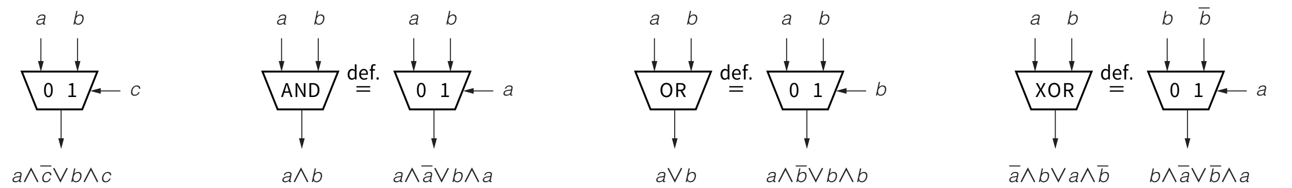
● 入力を変えてさまざまな回路を実現するセレクタ論理

論理機能を実現する回路は図2の基本回路だけではありません．例えば，トランスミッション・ゲート回路が挙げられます．この回路によってセレクタ論理を実現できます（図3）．

pMOSゲート（制御‘0’で信号通過．信号‘1’をよく通す）とnMOSゲート（制御‘1’で信号通過．信号‘0’をよく通す）を抱き合わせることにより，効率的な信号の遮断または通過を行います．図3(c)に示す等価回路からわかるように，トランスミッション・ゲート回路には抵抗成分があり，ゲートを通過するたびに電圧降下を招きます．pMOSまたはnMOSゲートの片方だけを利用することもあり，この場合を「パス・トランジスタ回路」と呼んで区別することがあります．

入力信号をa，b，制御信号をcとしたときのセレクタ論理$a \wedge \bar{c} \vee b \wedge c$を図3(a)の左半分で実現しています．また，図3(a)の右半分でその否定$\bar{a} \wedge \bar{c} \vee \bar{b} \wedge c$を同時に実現しています．このような実現法をデュアル・レール，またはダブル・レールといいます．回路規模などの制約により，実際にはシングル・レールで実現する場合が多いのですが，このときは否定信号発生回路を適当に追加します．また，電圧降下に対する対策としては，電圧レベルを回復するためにインバータ機能を持つ回路を挿入します．前後の論理を適当に等価変換して，NANDやNORなどを有効に使うこともできます．

セレクタ論理は強力です．図4に示すようにセレクタ論理だけでなく，入力状態を変えるだけでAND，OR，XOR



(a) 基本回路　(b) AND回路　(c) OR回路　(d) XOR回路

図4　セレクタ論理で表現できる基本ゲート論理

(a)のように，入力a，bを制御信号cでセレクトする回路がセレクタ回路の基本形．制御信号cをaに変えることでANDを，bに変えることでORを，そしてaをbに，bを$\overline{b}$に，cをaにすることでXORを実現できる．

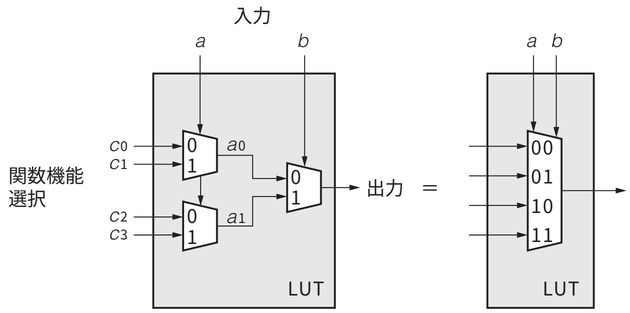


図5　FPGAで実現しているルックアップ・テーブルの基本構造

論理関数を実現するルックアップ・テーブルにもセレクタが活躍している．テーブルで表現される論理関数の種類とそれらを選択する機能がセレクタを組み合わせることにより実現できる．

などを簡単に実現できます．そればかりでなく，後述のセレクタ論理の合成能力を活用すれば，論理全体を簡単に表現できます．セレクタ論理は一般の論理合成ツールでは使うことができないので，人手で作成したものを使用します．

● FPGAもASICもセレクタを使った回路構成が基本

ところで，FPGAでは任意の論理を実現する基本回路構成は，セレクタを組み合わせたルックアップ・テーブルです(**図5**)．**図5**は，セレクタを2段重ねにした場合を示しています．セレクタの本来の制御側を入力a，bとして，本来の入力側を関数機能の選択部(c_0，c_1，c_2，c_3)として使用しています．

このルックアップ・テーブルでどのようなブール関数が実現できるか，**表1**で確かめてみましょう．**図5**の場合，$(2^2)^2=16$通りのブール関数があり，**表1**からこれらすべてを実現できていることがわかります．

表1　ルックアップ・テーブルで実現される論理の選択表

関数機能選択		データ入力		
		a		b
$c_0\ c_1$	$c_2\ c_3$	$a_0=c_0\wedge\overline{a}\vee c_1\wedge a$	$a_1=c_2\wedge\overline{a}\vee c_3\wedge a$	$a_0\wedge\overline{b}\vee a_1\wedge b$
00	00	0	0	0
00	01	0	a	$a\wedge b$ (AND)
00	10	0	$\overline{a}$	$\overline{a}\wedge b$
00	11	0	1	b
01	00	a	0	$a\wedge\overline{b}$
01	01	a	a	a
01	10	a	$\overline{a}$	$a\oplus b$ (XOR)
01	11	a	1	$a\vee b$ (OR)
10	00	$\overline{a}$	0	$\overline{a\vee b}$ (NOR)
10	01	$\overline{a}$	a	$\overline{a\oplus b}$ (XNOR)
10	10	$\overline{a}$	$\overline{a}$	$\overline{a}$ (NOTa)
10	11	$\overline{a}$	1	$\overline{a}\vee b$
11	00	1	0	$\overline{b}$ (NOTb)
11	01	1	a	$a\vee\overline{b}$
11	10	1	$\overline{a}$	$\overline{a\wedge b}$ (NAND)
11	11	1	1	1

XOR：$a\oplus b\equiv a\wedge\overline{b}\vee\overline{a}\wedge b$　XNOR：$\overline{a\oplus b}\equiv a\wedge b\vee\overline{a}\wedge\overline{b}$

次にASICの基本回路構成を考えてみます．例えば，**図6**に示すようにXNOR(排他的論理和の否定)fをCMOS回路で直接実現することを考えます．XNORの否定$\overline{f}$がXOR(排他的論理和)であることから，式(10)のように論理式を変形してやれば，**図6**の左側のように2段CMOSの回路で実現できます．

$$\left.\begin{aligned}\overline{f}&=a\wedge\overline{b}\vee\overline{a}\wedge b\\&=(a\vee b)\wedge(\overline{a}\vee\overline{b})\\&=(a\vee b)\wedge(\overline{a\wedge b})\end{aligned}\right\}\quad\cdots\cdots(10)$$

その双対回路(pMOSとnMOSの論理を交換する)として，**図6**の右側に示すXORも簡単に構成できます．CMOS回路の教科書はたいてい否定入力をインバータで得ているので見かけない回路だと思いますが，否定入力を使わなくても2段CMOS回路で実現できます．

● ラティス図を使って論理式を簡単化する

論理を簡単化するため，真理値を図示するカルノー図というものがよく使われますが，この図によって表現できる変数の数はせいぜい4～5個程度です．そこで，ここでは

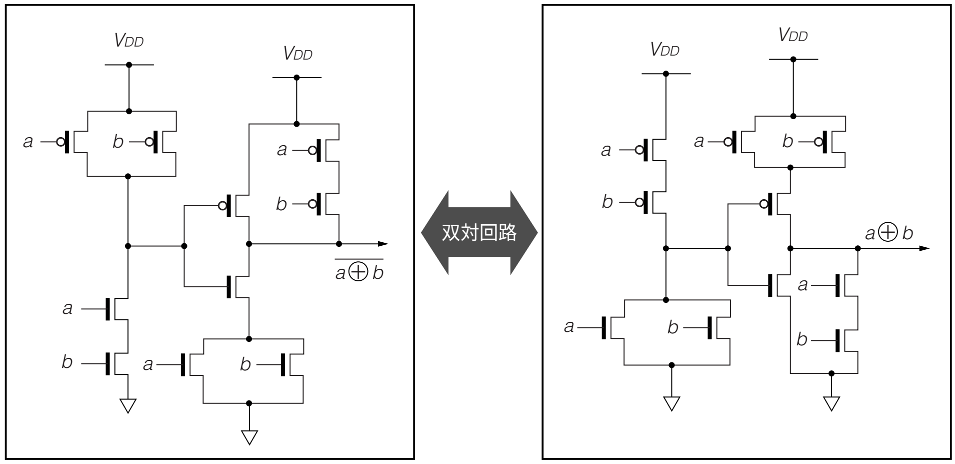


図6 CMOS回路で実現する排他的論理和回路

ASICでは，CMOS回路を複合的に組み合わせる論理合成によって設計する．その例として，排他的論理和の回路設計例を示す．

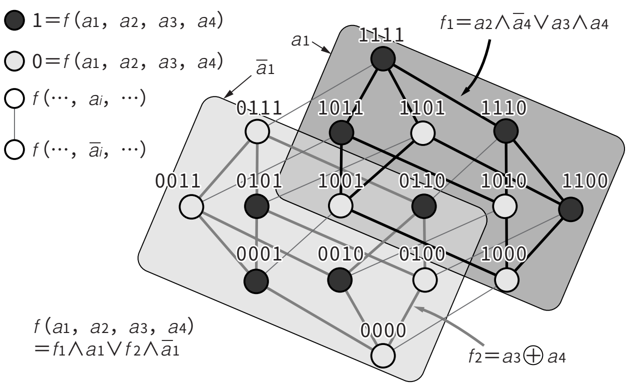


図7 ラティス図による論理の簡単化表現

論理関数fの真理値表に基づき，ラティス図の各入力変数値に対応するノード（○で示している箇所）について，その真理値が‘1’のときを濃い灰色，‘0’のときを薄い灰色と色分けする．同じ色のノード間に辺が存在すれば，それを表現している入力成分は無関係となり消せる．

原理的には何変数でもグラフ表現できるラティス図を紹介します．

図7に，4変数の場合のラティス図を示します．グラフのノード（○で示している部分）に対して色（**図7**では濃い灰色＝‘1’，薄い灰色＝‘0’）またはラベル値を付けることで論理関数fの真理値を与えています．n個の入力変数のある成分a_iについて，a_iの値が‘1’と‘0’（つまり，a_iと$\overline{a}_i$）の関係にあるときにノード間を辺で結びます．例えば，**図7**に示すように4入力変数a_1，a_2，a_3，a_4について4次元立方体のグラフ表現が得られます．任意のn変数の場合，表示した図がわかりにくくなってもコンピュータのデータ構造の中では，厳密に表現できます．特定の成分だけを射影成分として取り出し，表示，分析すればよいわけです．

図7には，セレクタの$f_1 = a_2 \wedge \overline{a}_4 \vee a_3 \wedge a_4$とXORの$f_2 = a_3 \oplus a_4$が埋め込まれています．そして，関数$f$は$f = f_1 \wedge a_1 \vee f_2 \wedge \overline{a}_1$のように，さらにセレクタでまとめられます．このように，ラティス図の射影グラフ表現において，セレクタ論理のパターンが見つかれば，セレクタ論理回路で簡単に実現できることがわかります．ここでは，これ以上詳しく説明しませんが，AND，OR，NOTだけで表現すると一見複雑そうな論理も，セレクタ論理表現を駆使することによって簡単化できるということを覚えておいてください．

3 カウンタ回路

図1に示したような筆算形式で論理積項を積み上げ表現する演算構成ができ上がったら，各けたにおいて‘1’の値を持つ積項の個数をカウントして，上位けたへけた上げを伝える処理を行わなければなりません．

● 複数入力のカウンタ回路はフル・アダーを組み合わせて

このカウントを行う基本回路（カウンタ回路）を**図8**に示します．**図8（a）**の回路は通常「フル・アダー」と呼ばれており，セレクタ論理2段で実現できる3入力のカウンタ回路です．一つ上位のけたへのけた上げc_{i+1}と現在のけたの結果s_iの二つの出力になることから，「3-2カウンタ」ともいいます．n入力のカウンタは，通常，3-2カウンタを組み合わせて構成します．

もっと効率良くn入力をカウントしたいときは，**図8（b）**に示す4-2コンプレッサ（セレクタ論理3段で実現）を用います．この回路では，4入力のうち，まず3入力を部分的にカウントすることで，けた上げc_{out}を途中で出力します．それから，3入力のカウント結果と4番目の入力，および

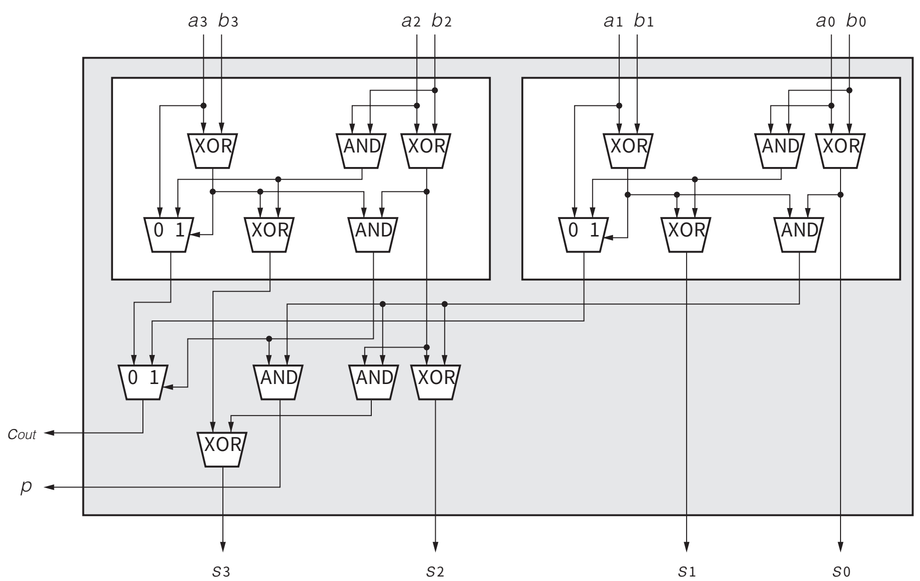
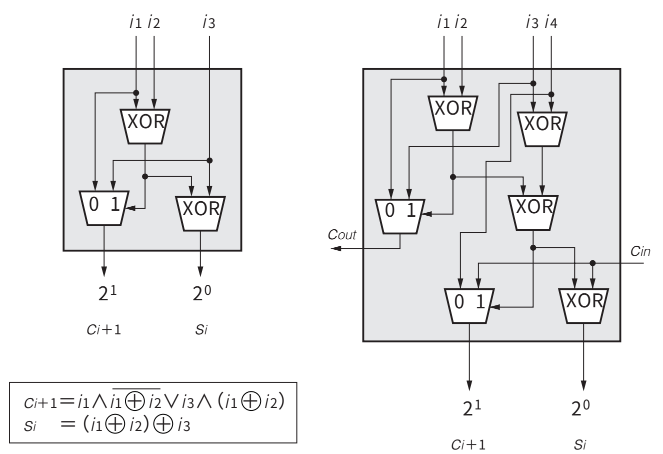


図9　4ビットCLA回路

CLAはいわばトーナメント試合．隣り合う各けたどうしのけた上げ合戦の勝ち残りを競うようなものである．けた上げは現在のけたの和が'1'のとき，下位のけたからのけた上げが来ると，さらに上位のけたへ伝える．CLA回路は，この状態を早い段階で予測しておく．

一つ下位のけたから途中のけた上げ入力(c_{in})を拾ってけた上げc_{i+1}を出し，現在のけたの和s_iを出力します．途中で変則的なけた上げの出し入れ(c_{out}，c_{in})があるので「4-2カウンタ(または5-3カウンタ)」と呼ぶのは変です．そこで，けたに関する入出力数の減少比が4：2であるという意味で，「4-2コンプレッサ」といいます．この回路がどのようにして導かれるのかを説明するのは難しいので，ここでは割愛します．みなさんは真理値表を作って，正しく動作する回路であることを確かめてください．

FPGAでは，3-2カウンタ・ベースで設計しやすいようにブロック構成がくふうされています．ただ，4-2コンプレッサについては現状では実現しにくいといえます．途中でけた上げの入出力があるからです．



(**a**) 3-2カウンタ(フル・アダー)　　(**b**) 4-2コンプレッサ

図8　カウンタを組み立てるための基本回路

(a)は3-2カウンタで，フル・アダーとも呼ばれる回路．(b)は4-2コンプレッサであり，2個の3-2カウンタを組み合わせている．ただし，入力i_1，i_2，i_3，i_4のけたの和が'0'のときは，i_1，i_2，i_3のけた上げはc_{out}で出力しているので，残るi_4の値をけた上げする．

● 早めにけた上げの状態を予測するCLA回路

カウンタでけた上げとけたの和の処理を構成しながら，段階的に各けたの出力数を減らしていくと，やがて2出力になる段階に至ります．下位のけたから順番にけた上げの完結処理を行い，最後に結果出力とすればよいわけですが，これではけた数に比例した時間がかかってしまいます．

そこで，けた上げ処理をトーナメント形式で加速するけた上げ先見加算器(CLA：carry look-ahead adder)を利用して，けた上げ伝播の完結処理を行います．**図9**に，基礎となる4ビットのCLA回路例を示します．あるけたの和が'1'になっている場合，下位のけたからのけた上げ入力が来たときに，上位へけた上げを伝えるためにこの状態を検出(**図9**のp信号を得る経路)しておきます．

4　2の補数表現を踏まえた演算器設計例

さて，演算器の具体的な設計例として，いろいろなところで使われる線形補間演算を取り上げて解説します．あまり知られていないのですが，線形補間演算では2の補数表現を巧みに利用することにより，効率的な演算器を構成できます．まず，符号なしデータの線形補間演算から説明します．

● 符号なしの場合は乗算器1個と同じ規模で実現

図10に示すように，両端点の値AとBが与えられたとき，その間の値CをD対$1-D$の割合で線形補間する場合，式(11)のように符号なしデータで表現します．

$$\left.\begin{aligned} A&=\left[a_{n-1}\cdots a_1a_0\right]\mathrm{b}=\sum_{j=0}^{n-1}a_j2^j \\ B&=\left[b_{n-1}\cdots b_1b_0\right]\mathrm{b}=\sum_{j=0}^{n-1}b_j2^j \\ D&=\left[0.d_{n-1}\cdots d_1d_0\right]\mathrm{b}=\sum_{j=0}^{n-1}d_i2^{-(n-i)}=2^{-n}\sum_{i=0}^{n-1}d_i2^i \end{aligned}\right\}\quad\cdots(11)$$

線形補間式は式(12)のように与えられるので，これに式(11)を代入します．そのとき，2の補数表現で負項を正項に変換する式(6)を用いて変形し，式をまとめ直します．そうすると，セレクタ論理に持ち込める形になります．

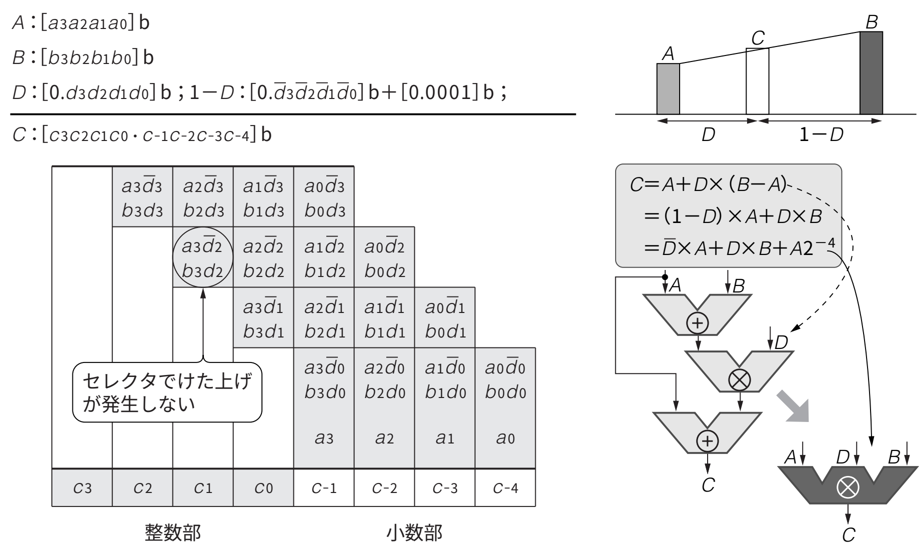


図10
2の補数表現による式の変形とセレクタ論理を用いた線形補間演算の効率的な実現

式(12)より，けた上げが発生しないように，セレクタ論理を使う形にして積項を書き並べる．結果として，整数部のみが必要の場合は小数部を適当に丸める(切り上げ，切り捨てを行う)．

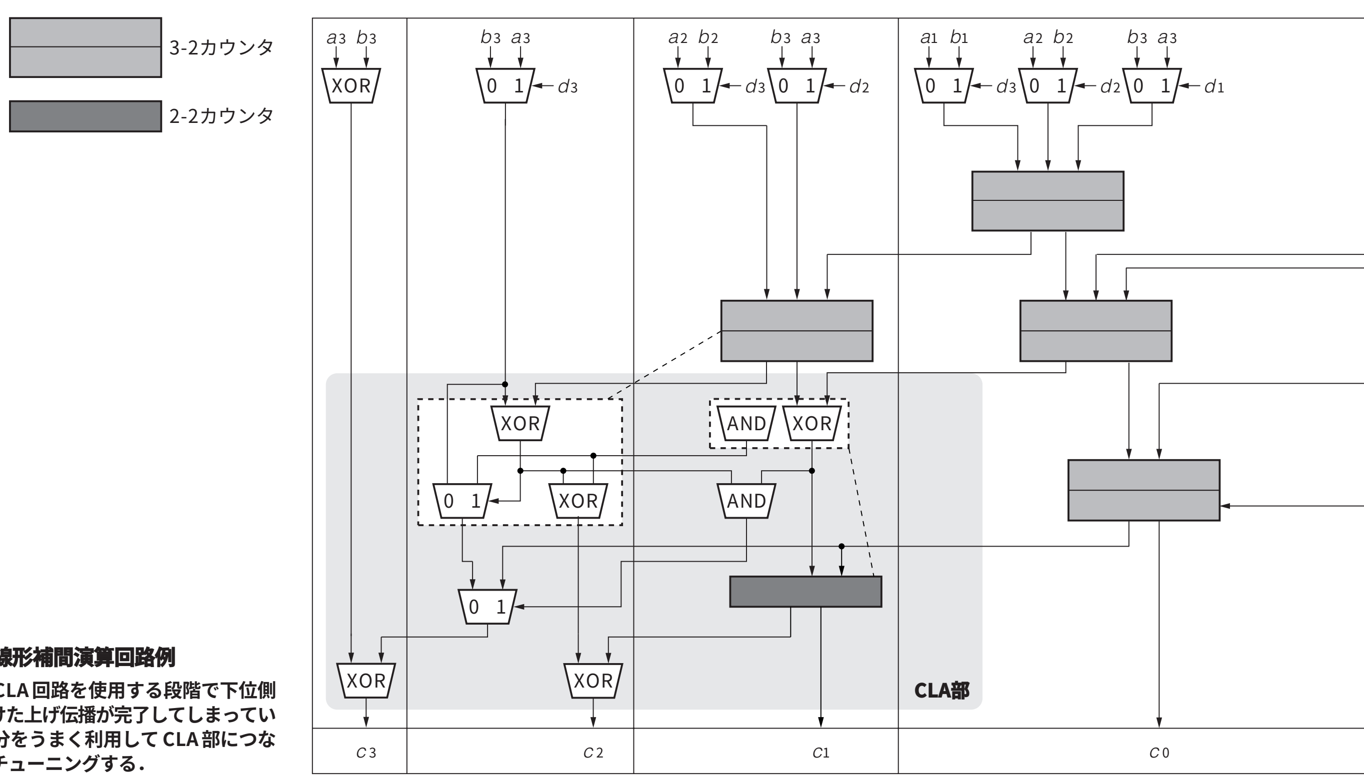


図12
4ビットの線形補間演算回路例

実際には，CLA回路を使用する段階で下位側のけたではけた上げ伝播が完了してしまっている．この部分をうまく利用してCLA部につなげるようにチューニングする．

$$
\begin{aligned}
C &= (1-D)A + DB \\
&= \left(2^n - 2^n + 1 + \sum_{i=0}^{n-1} \bar{d}_i 2^i\right) 2^{-n} \sum_{j=0}^{n-1} a_j 2^j + 2^{-n} \sum_{i=0}^{n-1} d_i 2^i \sum_{j=0}^{n-1} b_j 2^j \quad [1] \\
&= 2^{-n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_j \bar{d}_i 2^{i+j} + 2^{-n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} b_j d_i 2^{i+j} + 2^{-n} \sum_{j=0}^{n-1} a_j 2^j \\
&= 2^{-n} \left\{ \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(a_j \bar{d}_i + b_j d_i\right) 2^{i+j} \right\} \quad [2] \\
&\quad + 2^{-n} \sum_{j=0}^{n-1} a_j 2^j \\
&= \bar{D}A + DB + A2^{-n}
\end{aligned}
\quad \cdots\cdots\cdots (12)
$$

式(12)において，［1］の部分は2の補数表現を，［2］の部分はセレクタ論理を活用しています．

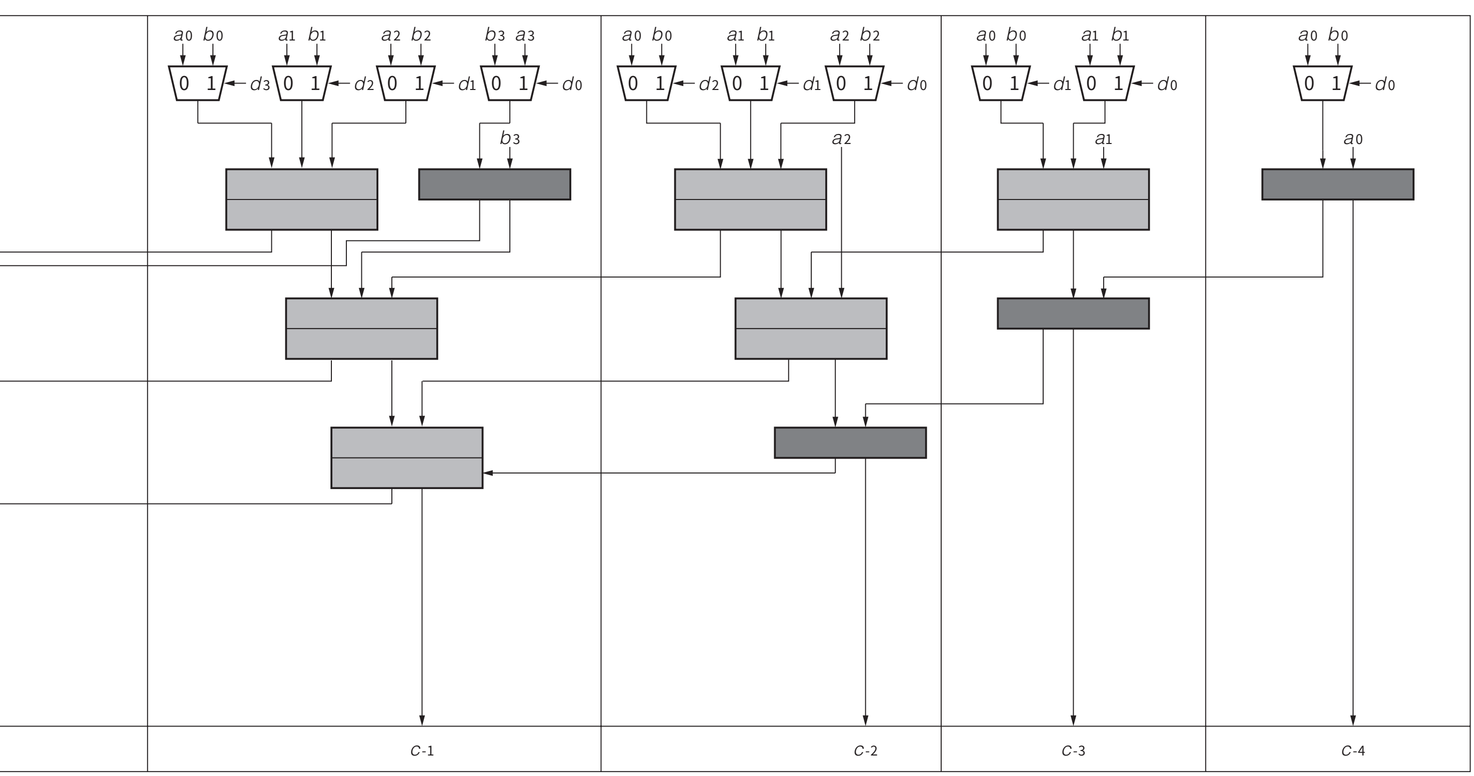
ここで注目すべきことは，セレクタ論理に持ち込んだけた加算においてd_iの値が‘0’のときはa_j，‘1’のときはb_jを

Column 「積和」と「和積」の違い

一般に，乗算器は回路規模も処理時間も大きくなるため，「乗算回数は少なければよい」と考える方もいるかもしれません．そこで，本文の**図10**に示したように，$C=A+D(B-A)$と式変形(和積演算)すれば乗算器一つになります．しかし，加算($B-A$)のけた上げ伝播が完了してからDと乗算することになり，かえって回路規模も処理時間も増えてしまいます．

一般に積和(乗算の後で加算)演算では，手間のかかるけた上げ伝播の完了は演算の最終段階でのみ行うため高速です．しかし，和積演算(加算の後で乗算)では最初に加算のけた上げ伝播を完了して，これを乗算の入力とします．また，演算の最終段階でもけた上げ伝播の完了を待たなくてはならないため，最初に手間のかかる加算という余分な処理があります．

現状のFPGAでは，本文の式(12)の積和演算で実現しようとすると，NOT演算もANDと同じくらい時間がかかってしまうので注意が必要です．**図10**の積項の積み上げを3-2カウンタで効率良く処理すれば，NOT演算の遅れを挽回でき，和積の式によって実現した回路よりも良い結果が出ます．



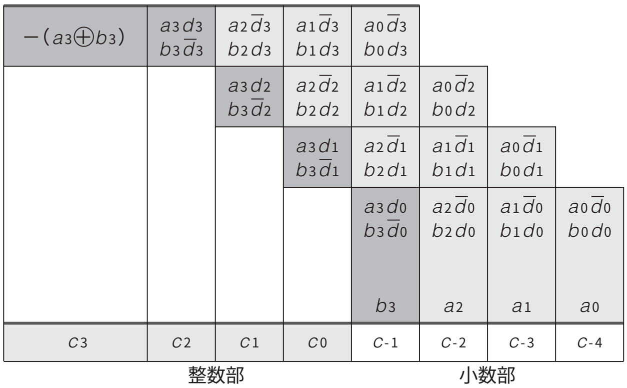


図11　符号付きデータの線形補間の扱いも簡単

符号なしデータの場合と異なるのは，符号部（濃い灰色の部分）のみ．くふうすれば符号なしデータの場合と共通化した演算回路を作ることも可能．

選択することになり，けた上げが生じないということです．そのため，**図10**に示すようにAの加算項が余分に加わるものの，乗算器1個とほぼ同じ規模で線形補間演算器を実現できます（p.47のコラム「『積和』と『和積』の違い」を参照）．

● 符号付きの場合も簡単に線形補間できる

今度は符号付きデータの場合について，線形補間を見てみましょう．符号付きデータA，Bは，式(13)で与えられます．

$$\left.\begin{aligned}A&=\left[a_{n-1}\cdots a_1a_0\right]b=-a_{n-1}2^{n-1}+\sum_{j=0}^{n-2}a_j2^j\\B&=\left[b_{n-1}\cdots b_1b_0\right]b=-b_{n-1}2^{n-1}+\sum_{j=0}^{n-2}b_j2^j\\D&=\left[0.d_{n-1}\cdots d_1d_0\right]b=\sum_{i=0}^{n-1}d_i2^{-(n-i)}\end{aligned}\right\}\quad\cdots\cdots\cdots(13)$$

線形補間の式に代入して展開し，正項に変換して式をまとめ直すと，式(14)が得られます．

$$\begin{aligned}C&=\left(1-\sum_{i=0}^{n-1}d_i2^{-(n-i)}\right)\left(-a_{n-1}2^{n-1}+\sum_{j=0}^{n-2}a_j2^j\right)\\&\quad+\left(\sum_{i=0}^{n-1}d_i2^{-(n-i)}\right)\left(-b_{n-1}2^{n-1}+\sum_{j=0}^{n-2}b_j2^j\right)\\&=\left(-a_{n-1}2^{n-1}+a_{n-1}\sum_{i=0}^{n-1}d_i2^{i-1}\right)\\&\quad+\left(2^{-n}+\sum_{i=0}^{n-1}\bar{d}_i2^{-(n-i)}\right)\left(\sum_{j=0}^{n-2}a_j2^j\right)\qquad[1]\\&\quad-b_{n-1}2^{n-1}+b_{n-1}2^{-1}+b_{n-1}\sum_{i=0}^{n-1}\bar{d}_i2^{i-1}\\&\quad+\left(\sum_{i=0}^{n-1}d_i2^{-(n-i)}\right)\left(\sum_{j=0}^{n-2}b_j2^j\right)\\&=-\left(a_{n-1}+b_{n-1}\right)2^{n-1}+b_{n-1}2^{-1}\\&\quad+\left(a_{n-1}\sum_{i=0}^{n-1}d_i2^{i-1}+b_{n-1}\sum_{i=0}^{n-1}\bar{d}_i2^{i-1}\right)\\&\quad+\left(2^{-n}+\sum_{i=0}^{n-1}\bar{d}_i2^{-(n-i)}\right)\left(\sum_{j=0}^{n-2}a_j2^j\right)\qquad[2]\\&\quad+\left(\sum_{i=0}^{n-1}d_i2^{-(n-i)}\right)\left(\sum_{j=0}^{n-2}b_j2^j\right)\end{aligned}\quad\cdots\cdots(14)$$

式(14)において，[1]の部分で正項変換を行い，[2]の部分で式をまとめています．

ここで，式(14)の最上位けたの符号部は$-(a_{n-1}+b_{n-1})2^{n-1}$です．$a_{n-1}=b_{n-1}=1$のときは，式の形からわかるように下位のけたからかならずけた上げ入力があります．そのため，符号部は$(-2+1)2^{n-1}$となり，論理的には$-(a_{n-1}\oplus b_{n-1})2^{n-1}$としておけば，$(-2+1)2^{n-1}$の場合の回路も不要になり，結果としてつじつまが合うことになります（**図11**）．

最後にまとめとして，符号付き4ビットの線形補間演算器の回路設計例を**図12**（pp.46-47）に示します．回路性能のおおよその見積もりを行いやすいように，セレクタ論理で数えた段数でそろえて表示しています．3-2カウンタ（一部2入力2出力の2-2カウンタ）で各けたの‘1’の数をカウントしながらけた上げ伝播を行うとともに，段階的に各けたの和の出力数を減らしていきます．そして，各けたの出力数が2個以内にそろったところで，CLA回路で加算します．ここでは，一般的なCLA回路を基本にして回路をチューニングしています．

とのむら・もとのぶ
大日本印刷（株）電子デバイス事業部 電子デバイス研究所

<著者プロフィール>
外村元伸．20年以上勤務した日立製作所 中央研究所を4年前に退職し，ベンチャ企業などを経て，現職．少年時代，豆電球を使って数字のかたちで足し算を直接表示することを思いついたが，スイッチ配線が複雑になって何とかならないかと考えていた．「トランジスタ技術」が創刊されたばかりのころ，トランジスタで演算器を作る記事があり，夢中で読んで自作しようとしたが，当時のこづかいで部品が買えるわけがなかった．そのとき，2進法を知って感激した．

編集部注：本稿の続編は，本誌の次号以降に連載記事として掲載する予定です．